AN3140
Application note

How to configure the SPEAr3xx general purpose timers (GPTs)

Introduction

This application note provides information about how to configure the general purpose timers (GPTs) integrated in the SPEAr3xx embedded MPU family.

General purpose timers (GPTs) play an important role in any system as they provide a means of calculating time for controlling the execution of various operations. In case of an operating system, they are used for the system tick generation, usually every 10 ms; in other applications they can be used to get a finer granularity for controlling the timing of events.

The purpose of this application note is to explain how to read the free running timer counter and configure the clock source of the various GPTs that are integrated in the SPEAr3xx architecture. It also describes and proposes a solution for the problem reported during the Puppy Linux project concerning the status register interrupt bit clear issue.

Contents

1 General purpose timers (GPTs) in SPEAr3xx 3

2 Reading a free-running timer counter 5

3 Scenario with slow CNT_Clk and fast READ_Clk 7

4 How to configure CNT_Clk and READ_Clk to be synchronous 8

5 3-read software workaround 9

6 Status register interrupt bit clear issue 10

6.1 Problem description 10

6.2 Proposed solution 11

7 Summary 12

8 Revision history 13

1 General purpose timers (GPTs) in SPEAr3xx

In the SPEAr3xx architecture, there are three different GPT blocks located in the various subsystems. Each timer block consists of two independent channels, each one with a 16-bit counter register.

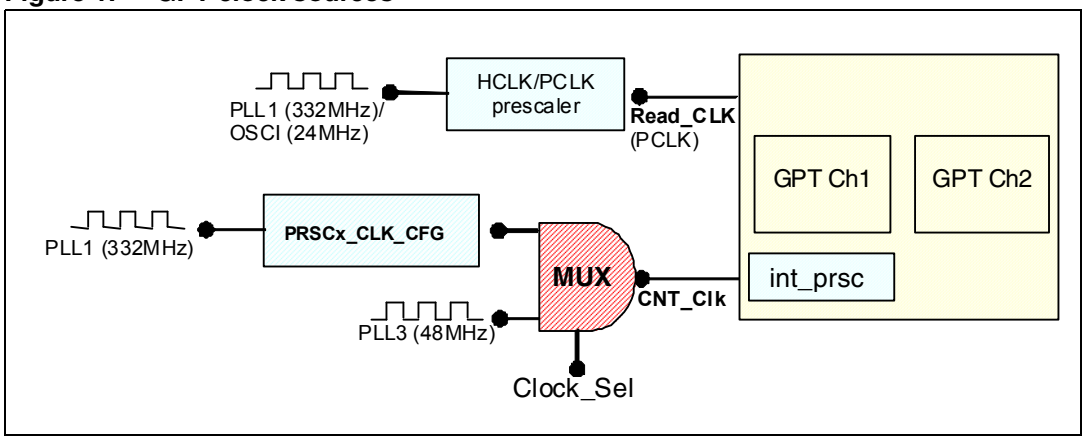
Table 1. GPTs in SPEAr3xx

	Subsystem	Base address
GPT1	ARM	0xF000_0000
GPT2	Basic1	0xFC80_0000
GPT3	Basic2	0xFCB0_0000

Each timer has a *READ_Clk*, input which is the APB clock (PCLK), and a *CNT_Clk*, which can be selected by the user from a list of clock sources.

- *READ_Clk* (PCLK): When SPEAr3xx is in *normal* mode, it takes the input from PLL1 divided by a programmable prescaler, whose reset values impose the ratio 1:2:4 to the core_clk, HCLK and PCLK clocks. When SPEAr3xx is in *slow* mode, it takes directly the input from the OSCI signal.
- *CNT_Clk*: The clock source can be selected as either a fixed 48 MHz or the PLL1 itself divided by a programmable prescaler, which is defined in the *PRSC1_CLK_CFG* register (0xFCA8_0044) for GPT1, *PRSC2_CLK_CFG* register (0xFCA8_0048) for GPT2 and *PRSC3_CLK_CFG* register (0xFCA8_004C) for GPT3. The *CNT_Clk* may then be further divided by a GPT internal 4-bit prescaler able to divide up to 256 times ('/256').

Figure 1. GPT clock sources



The following table describes the clock selectors (Clock_Sel) for each GPT.

Table 2. GPTx clock source selector

	Register	Address	Value
GPT1	PRPH_CLK_CFG [08]	0xFCA8_0028 (bit8)	0: PLL3 48 MHz 1: PLL1 (*PRSC1_CLK_CFG*)
GPT2	PRPH_CLK_CFG[11]	0xFCA8_0028 (bit11)	0: PLL3 48 MHz 1: PLL1 (*PRSC2_CLK_CFG*)
GPT3	PRPH_CLK_CFG[12]	0xFCA8_0028 (bit12)	0: PLL3 48 MHz 1: PLL1 (*PRSC3_CLK_CFG*)

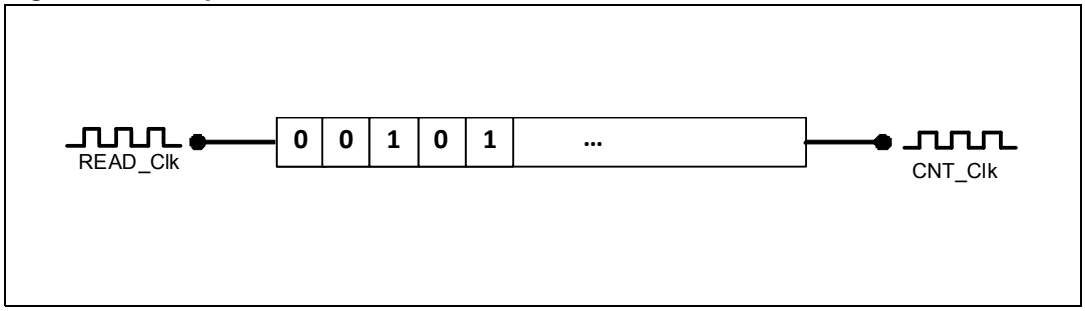
The SPEAr3xx GPTs always generate precise alarm interrupts, for example in the case of a system tick for a RTOS. Nevertheless, as you can see in *Section 2: Reading a free-running timer counter*, GPTs can return unpredictable read values when they are running and the input clock is asynchronous (or not in phase).

2 Reading a free-running timer counter

When the GPT interrupt is enabled, the interrupts generated at each timer wrap-around condition are always triggered at the right frequency, however reading the timer counter when the timer itself is active and free-running may present some difficulties which are described below.

In a simplified scenario, a hardware timer block can be seen just as a simple counter register with two input clocks: *CNT_Clk* for incrementing/decrementing the counter and *READ_Clk* for synchronizing the READ accesses of the bus the timer is connected to.

Figure 2. Simplified timer

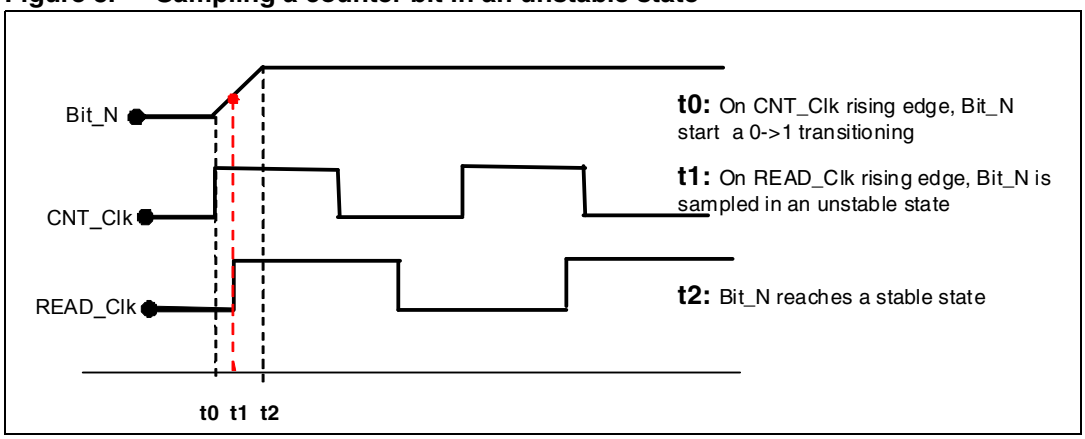


The two clocks can be either synchronous, coming from the same source PCLK, or completely asynchronous, for example coming from two different sources.

When the two clocks involved in the scenario are asynchronous, then the value retrieved by the CPU in a read counter operation is unpredictable, and might be completely different from the real value in the register.

The situation is due to the fact that the *READ_Clk* is sampling the counter bits while they are in a transitioning, unstable phase.

Figure 3. Sampling a counter bit in an unstable state


Bit_N
CNT_Clk
READ_Clk
t0 t1 t2
t0: On CNT_Clk rising edge, Bit_N start a 0->1 transitioning
t1: On READ_Clk rising edge, Bit_N is sampled in an unstable state
t2: Bit_N reaches a stable state

The above scenario may take place during any kind of transition (0->1 or 1->0) and for any bit in the register.

If one of the bits impacted has a large weight (significant position) in the counter, then the difference between the value returned in the read transaction and the real value of the counter can be very large.

Let's take as an example a counting down 16-bit counter transitioning from the value **1000_0000_0000_0000** (0x8000) to **0111_1111_1111_1111** (0x7FFF). Since the transition time of the 16 bits can be slightly different between each other, then the 16-bit counter value could be read by the CPU randomly as 0x0000 or 0xFFFF leading to a big difference from its real value.

A similar scenario may also occur in case the two clocks are synchronous, but not in phase. In this case, in fact, the *READ_Clk* may sample the bit during its unstable state period.

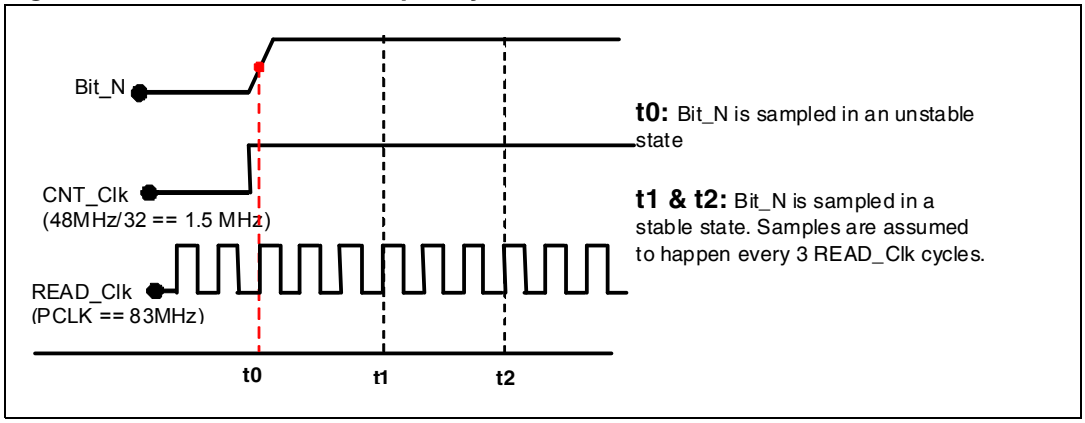
So, the two clocks must be synchronous and in phase.

3 Scenario with slow CNT_Clk and fast READ_Clk

In certain cases, for example when the timer is used by an operating system to generate the system tick, the *CNT_Clk* (after prescaling) is usually much slower than *READ_Clk*. For example, let's suppose you need to generate a tick every 10 ms; the GPT with a clock source of 48 MHz might be programmed using a '/32' internal prescaler and a counter equal to 15000.

This results in a great number (around 60) *READ_Clk* '*sampling cycles*' for every single *CNT_Clk* cycle. Or, in other words, *CNT_Clk* is about 60 times slower than *READ_Clk*.

Figure 4. CNT_Clk at low frequency


Bit_N
CNT_Clk
(48MHz/32 == 1.5 MHz)
READ_Clk
(PCLK == 83MHz)
t0
t1
t2
t0: Bit_N is sampled in an unstable state
t1 & t2: Bit_N is sampled in a stable state. Samples are assumed to happen every 3 READ_Clk cycles.

Let's see what happens if the CPU does three consecutive read operations instead of a single one. Since the bit instability lasts much less than the *CNT_Clk* time period, we can say that, out of 3 *READ_Clk* edges, only one will ever fall into the bit instability window. The other two are stable.

Moreover, since *CNT_Clk* is about 60 times slower than *READ_Clk,* the two stable read operations return counter values that differ by 1 in the worst case, which is when there is a *CNT_Clk* rising edge between the first and third read operations. Of course, interrupts should be disabled during the reads.

So, reading three times the counter and discharging the unstable value (if any) is a valid workaround that can be used for all GPTs of SPEAr3xx in similar scenarios.

In general, this workaround is valid when the minimum period of *CNT_Clk* is greater than 3 times the read_cycle_time. The read_cycle_time depends on the CPU frequency, and also on the way the reads are implemented, so they should be carefully evaluated.

4 How to configure CNT_Clk and READ_Clk to be synchronous

This method, which is very simple to implement, should work for **all GPTs**.

The most common configuration is when SPEAr3xx is in normal mode with system clocks fed by PLL1. In case the system is set in this mode, you can just select PLL1 as CNT_Clk to guarantee the synchronicity between CNT_Clk and READ_Clk.

To set the input clock source of GPTx to PLL1 you need to use PRPH_CLK_CFG register (0xFCA8_0028). There are three different bits, one for each GPT block.

- For **GPT1**: PRPH_CLK_CFG [8] = 1
- For **GPT2**: PRPH_CLK_CFG [11] = 1
- For **GPT3**: PRPH_CLK_CFG [12] = 1

In case SPEAr3xx enters the slow mode, for example to save power after detecting a period of inactivity, the HCLK/PCLK system clocks are directly fed from the OSCI at 30 MHz. In this mode READ_Clk (OSCI) and CNT_Clk (PLL1) become asynchronous again.

5 3-read software workaround

Below you can see a proposal for a software workaround that works for the scenario described in the previous section. This workaround works well in case of the previously described scenario, where READ_Clk is much faster than CNT_Clk, as well as when they have similar frequencies.

In the first case the MAX_DIFF should be defined as ‘1’, while in the second case (similar frequencies) a higher value should be selected. This value should be fine tuned depending to the two real frequencies.

Example code

```
/*
 * The following routine implents the 3-read workaround.
 * MAX_DIFF equals to 5 should work in case of READ_Clk==75MHz and
 * CNT_Clk==48MHz.
 */
#define MAX_DIFF 5

bool Timer_Read_Workaround(UINT32 *valid_timer_cnt_value)
{

   UINT32 timer_value1, timer_value2, timer_value3;
   UINT32 valid_timer_cnt_value;

   /* To avoid any interrupt that might delay the reading. */
   DISABLE_ALL_INTERRUPT;

   timer_value1 =  READ_TIMER_CNT();
   timer_value2 =  READ_TIMER_CNT();
   timer_value3 =  READ_TIMER_CNT();

   ENABLE_ALL_INTERRUPT;

   if ((timer_value2 - timer_value1) <= MAX_DIFF) {
      *valid_timer_cnt_value = timer_value2;
   }
   else if ((timer_value3 - timer_value1) <= MAX_DIFF) {
      *valid_timer_cnt_value = timer_value3;
   }
   else if ((timer_value3 - timer_value2) <= MAX_DIFF) {
      *valid_timer_cnt_value = timer_value2;
   }
   else
      return FALSE;

   return TRUE;
}
```

6 Status register interrupt bit clear issue

This section provides a technical explanation of the problem reported during the Puppy Linux project debugging about the usage of the GPT and suggests a safe solution for it.

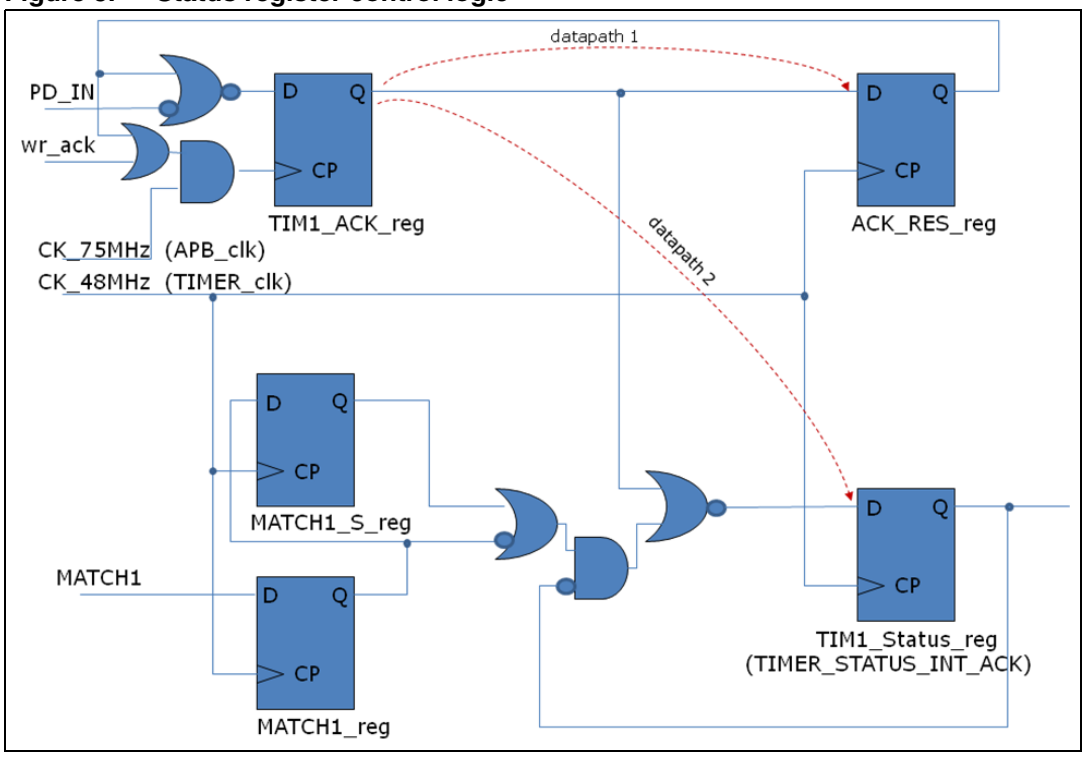
6.1 Problem description

During the usage of the GPT inside SPEAr3xx an issue can be faced when the timer is used in AutoReload mode. For more details on GPT usage, please refer to SPEAr3xx user manual.

The unexpected behavior might occur when the CPU tries to clear the interrupt request bit inside the status register of the timer (TIMER_STATUS_INT_ACK). This clear operation is performed through a write '1' command from the CPU to the APB interface of the GPT. It could happen that this write fails, the interrupt request bit is not cleared and the GPT is not able to generate further interrupts until a new successfully write '1' operation is performed. The risk in this case, as the GPT is in AutoReload mode, is to lose some interrupt events.

The source of this weakness is ~~in~~ the asynchronism between the APB_clk (75 MHz in the Puppy Linux application) of the write command and the Timer_clk (48MHz in Puppy Linux application) of the status register. To better understand the mechanism of this interrupt clear operation, please refer to *Figure 5*.

Figure 5. Status register control logic



When a match condition is reported, the structure of the MATCH1_reg and MATCH1_S_reg generates a pulse 0 to 1 to 0, which lasts one TIMER_clk cycle, on the input D of the

TIMER_STATUS_INT_ACK register. This '1' is synchronously captured by TIMER_STATUS_INT_ACK and the interrupt request is generated.

The status '1' is permanently kept inside the register by the feedback structure, until the interrupt clear operation is performed. The clear operation consists in a write '1' on the TIM1_ACK_reg through PD_IN input. The output of TIM1_ACK_reg goes to the TIMER_STATUS_INT_ACK register and clears it.

The feedback structure between TIM1_ACK_reg and ACK_RES_reg ensures that the output '1' on TIM1_ACK_reg is kept till it is not properly captured by ACK_RES_reg. If ACK_RES_reg and TIMER_STATUS_INT_ACK reg are balanced in terms of clock skew (less than 2 ps in wc), the proper capturing of ACK_RES_reg will guarantee the proper capturing of TIMER_STATUS_INT_ACK.

This structure is supposed to limit the effect of the lack of synchronization between the two clocks, but it still has one limit: the metastability.

When the data arrives to the FF input pins of both ACK_RES_reg and TIMER_STATUS_INT_ACK reg simultaneously with TIMER_clk, the behavior of the FF is not predictable. The only thing we can guarantee is that after 1-2 ns the FF goes to a stable value but this value is unpredictable.

The static timing analysis on the two registers showed that the datapath 1 on the ACK_RES_reg is slightly faster than the datapath 2 on the TIMER_STATUS_INT_ACK register. This means that the ACK_REG_reg has higher chances to properly capture the correct values in the metastability windows.

Only in this specific situation, for example when ACK_RES_reg captures '1' while TIMER_STATUS_INT_ACK misses the capture, the issue is present because ACK_RES_reg drives TIM1_ACK_reg to '0', definitively preventing TIMER_STATUS_INT_ACK from getting cleared.

Assuming that the critical event is when the TIMER_clk phase is equal to the APB_clk phase + datapath 2, it is possible to estimate the occurrence of this event.

Within a period of 16 TIMER_clk cycles (or equivalently 25 APB_clk cycles) the two clocks get realigned. Within this "periodical window" the phase differences of the two clock edges change from 0 to 13.3 ns (75 MHz period) with a granularity of about 833 ps. Considering the metastability window of less than 300 ps (FF setup+hold requirement), we can state that the critical event can happen only once within this window, if for example the occurrence is 1/16.

6.2 Proposed solution

The proposed solution is simple. Two successive write '1' operations guarantee that one of the two writes is successful. The atomic sequence of the two operations is mandatory, no further operation can occur between. Knowing that each write operation takes 3 APB_clk cycles, this double write operation ensures that both writes occur in a single "periodical window" (16 TIMER_clk or 25 APB_clk).

About current Puppy Linux solution (8 successive write '1' operations), this is not critical at all: once a write '1' succeeds, the next write '1' operations are not sensed at TIMER_STATUS_INT_ACK thanks to its feedback structure, so no risk of metastability can further happen.

7 Summary

A general purpose timer can be seen as a simple counter with two clocks in input: *READ_Clk* (for the slave interface) and *CNT_Clk* (for incrementing/decrementing the counter).

The *CNT_Clk* for the GPT in SPEAr3xx can be selected between a fixed 48 MHz source and PLL1, which is also the source clock for the rest of the system. The *READ_Clk* is derived from PLL1 in normal mode (PCLK) and from the 24 MHz OSCI in slow mode.

Having a fixed clock source different from the system clock has the advantage of eliminating the need for reconfiguring the GPT registers if the system clock frequency is slowed down. However, it introduces the possibility of obtaining an unpredictable result when reading the timer value, due to the non-synchronous operation of the two clocks.

8 Revision history

Table 3. Document revision history

Date	Revision	Changes
03-May-2010	1	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2010 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com